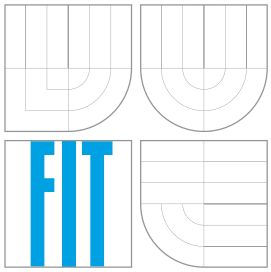
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

PARALELIZACE SESTAVENÍ V PROSTŘEDÍ JENKINS

BUILD PARALLELIZATION IN JENKINS ENVIRONMENT

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MICHAELA LUKÁŠOVÁ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. LUDĚK DOLÍHAL

BRNO 2016

Zadání bakalářské práce/18448/2015/xlukas09

Vysoké učení technické v Brně - Fakulta informačních technologií

Výzkumné centrum informačních technologií Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Lukášová Michaela**
Obor: Informační technologie
Téma: **Paralelizace sestavení v prostředí Jenkins**
Build Parallelization in Jenkins Environmnent
Kategorie: Analýza a testování softwaru

Pokyny:

1. Seznamte se s aktuálním systémem vytváření release.
2. Nastudujte možnosti paralelizace v prostředí Jenkins.
3. Dle pokynů vedoucího navrhněte systém, který umožní provádět paralelní sestavení balíku. Zejména se soustřeďte na urychlení.
4. Dle návrhu implementujte.
5. Zhodnoťte řešení z hlediska hardwarové náročnosti a rychlosti.

Literatura:

- Codasip. *Codasip Studio User Guide*. Codasip s.r.o., 2015
- John Ferguson Smart. Jenkins the Definitive Guide, O'Reilly Media, 2011

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Dolíhal Luděk, Ing.**, VCIT FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 18. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Výzkumné centrum informačních technologií
Božetěchova 2, 612 66 Brno
L.S.

prof. Ing. Tomáš Hruška, CSc.
vedoucí ústavu

Abstrakt

Tato práce se zabývá paralelním sestavením balíku vývojového prostředí Codasip Studio. Zaměřuje se na možnosti paralelizace v prostředí Jenkins. Implementované řešení se soustřeďuje zejména na urychlení aktuálního procesu sestavení. Řešení využívá několika zásuvných modulů prostředí Jenkins a také několika shellových skriptů zajišťujících spuštění překladu, instalace či tvorby výsledného balíku.

Abstract

The goal of this bachelor's thesis is parallelization of building Codasip Studio, highly automated developement environment. It focuses on parallelization in Jenkins environment. The implemented solution is mainly focused on speeding up the actual build process. The solution uses a number of Jenkins plugins and several shell scripts, which ensures start of compilation, installation or creation of the final package.

Klíčová slova

Paralelizace, prostředí Jenkins, distribuované zpracování, Master/Slave architektura, kompilace, instalace, artefakty, dotazování, zásuvný modul

Keywords

Parallelization, Jenkins environment, distributed processing, Master/Slave architecture, compilation, installation, artefacts, polling, plugin

Citace

LUKÁŠOVÁ, Michaela. *Paralelizace sestavení v prostředí Jenkins.* Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Dolíhal Luděk.

Paralelizace sestavení v prostředí Jenkins

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně pod vedením pana Ing. Luďka Dolíhala. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

.....
Michaela Lukášová
17. května 2016

Poděkování

Tímto bych chtěla poděkovat vedoucímu práce Ing. Luďkovi Dolíhalovi za jeho odbornou pomoc a nasměrování na správně řešení.

Obsah

Kapitola 1

Úvod

Vývoj softwarových produktů bývá doprovázen neustálými změnami z hlediska zdrojových kódů. Změny mohou být důsledkem opravování nalezených chyb, refaktorizací kódu při potřebě jeho zjednodušení a optimalizaci, ale také například realizací nové funkcionality softwaru. S přibývající funkčností a narůstajícím počtem řádků kódu dochází ke zvětšování jeho velikosti. Proces sestavení výsledného produktu se tak mimojiné stává náročnějším na výpočetní výkon procesorů. Nákladnost zvyšování počtu procesorů však není úměrná neustále se zvyšujícím nárokům na výkon, jež mohou procesory poskytnout.

Východiskem této situace se stalo zavedení možnosti paralelizace obtížných výpočtů. Rozdělení výpočtů na menší části a jejich souběžné zpracování umožnilo snížit zátež jednotlivých procesorů a poskytnout celkové zrychlení sestavení výsledného produktu.

I pro velké projekty je potřeba dodávat zákazníkům či vývojářum aktuální verze softwaru v určitých intervalech. Ne vždy jsou však tyto intervaly uspokojující. Firma Codasip vydává stabilní verzi balíku vývojového prostředí Codasip Studio přibližně jednou za měsíc pro zákazníky, verze pro tým vývojářů je vydávána jedenkrát za den. Pro tento tým by však bylo vhodnější, kdyby se proces vydávání mohl opakovat i několikrát denně. To by umožnilo nové verze řádně otestovat a vydat další balík s opravenými chybami již v průběhu několika hodin. Další výhodou by také byla možnost vydávat v jednom dni jak novou tak i starou verzi balíku, vzhledem k podpoře zpětné kompatibility. Proces sestavení balíku však dosahuje na některých systémech doby běhu až okolo 2–3 hodin. Pokud vezmeme v úvahu, že se navíc vytvářejí balíky ve dvou stupních licenční ochrany, zdvojnásobuje se tak čas běhu až na interval 4–6 hodin.

Cílem této práce bude pokusit se snížit dobu sestavení balíku Codasip Studio. Bude nutné se blíže zaměřit na proces sestavení v prostředí Jenkins a nalézt možnosti, jak tento proces rozdělit na menší části a ty se pokusit paralelizovat.

V této práci se nejdříve v kapitole číslo 2 seznámíme s vývojovým prostředím Codasip Studio, řekneme si něco o jeho využití, popíšeme si organizaci vrstev, ze kterých je složeno a uvedeme si také několik nástrojů, které toto prostředí poskytuje. Kapitola číslo 3 se zabývá samotným pojmem paralelizace a principy paralelních a distribuovaných výpočtů, které nás uvedou blíže do problematiky souběžného zpracování, jehož implementace je cílem této práce. Čtvrtá kapitola obsahuje popis prostředí Jenkins a jeho využití v integračním procesu vývoje prostředí Codasip Studio. Další kapitola blíže popisuje aktuální stav sestavování balíku (kapitola číslo 5). V posledních dvou kapitolách je poté vytvořen návrh paralelizace sestavení a jeho následná implementace, zakončená shrnutím dosažených výsledků.

Kapitola 2

Codasip Studio

Firma Codasip vznikla z výzkumného projektu Lissom, zahájeného v roce 2004 na Fakultě informačních technologií Vysokého učení technického v Brně. Tento projekt se zaměřoval na oblast vývoje jazyka pro popis architektury víceprocesorového systému na čipu a na transformaci tohoto popisu do pokročilých softwarových nástrojů nebo hardwarové realizace. V roce 2006 vznikla firma Codasip jako firemní spin-off z Aps Brno.

Codasip poskytuje vývojové prostředí, zvané Codasip Studio, pro tvorbu aplikačně specifických instrukčních procesorů (ASIP) založeném na otevřených standardech. Tyto procesory jsou navrženy tak, aby mohly sloužit ke konkrétním, úzce zaměřeným účelům. Dokáží tím zajistit nižší spotřebu a vyšší efektivitu.

Firma vyvíjí také vlastní vzorové procesory pro využití v procesorových systémech. Poskytuje vysoce strukturovaný, hierarchický jazyk pro popis architektury zvaný CodAL a také Codasip IDE. Jedná se o grafické uživatelské prostředí, založené na Eclipse, které mimojiné podporuje i generování a spouštění veškerých Codasip nástrojů (viz sekce 2.3).

2.1 Využití Codasip Studia

Vývojové prostředí Codasip Studio podporuje kromě automatizovaného vývoje ASIPů také verifikaci, programování či debugování. K návrhu ASIPů je využíván již zmíněný jazyk CodAL [1].

CodAL se syntaxí podobá jazyku C a umožňuje popsat široké množství typů procesorů, jako například RISC, VLIW, SISC či DSP. Z tohoto popisu, následně zpracovaného do formátu XML, je možné generovat vývojové a ladící prostředky, jako je například assembler, simulátor, profiler nebo C/C++ překladač, založený na LLVM. S použitím tohoto jazyka byly realizovany například architektury MIPS, RISC-V, x86 a jiné.

Mezi další použití se řadí modelování nového mikrokontroléru, také je možné modelovat již existující architektury s cílem nahrazovat rychle zastarající nástroje. Je kladen důraz na optimalizací již existujicich jader pomocí rozšíření instrukční sady (Instruction Set Extensions – ISE).

Codasip Studio umožňuje také generování syntetizovatelného popisu procesoru a jeho následnou funkční verifikaci. Zahrnuje i podporu pro multiprocesorové profilování, programování, verifikaci a debugování.

2.2 Organizace vývojového prostředí

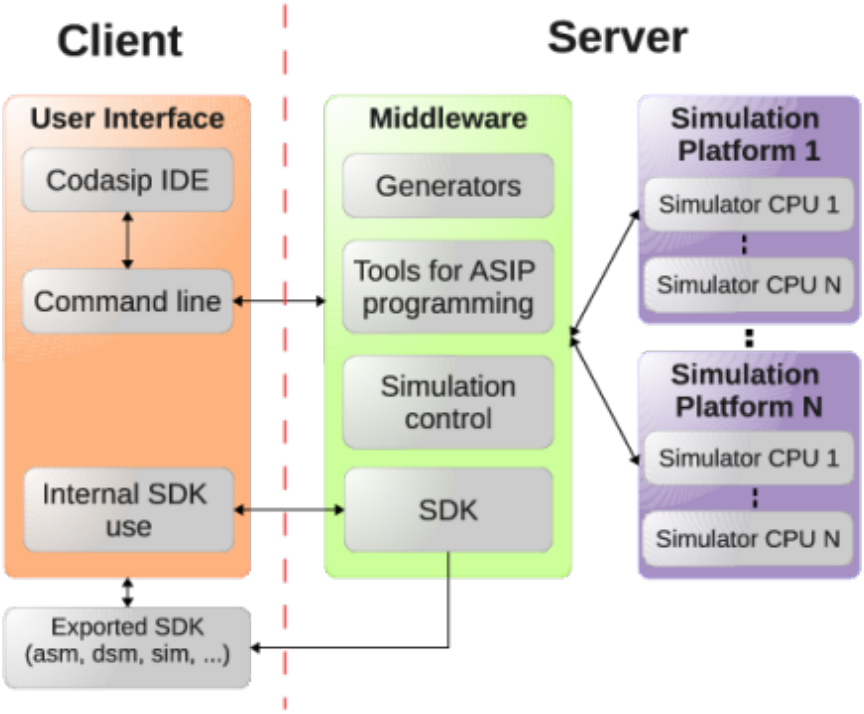
Codasip Studio se skládá ze 3 vrstev – vrstvy uživatelského rozhranní, middlewaru a vrstvy simulační [2]. Komunikace mezi nimi je realizována pomocí protokolu TCP/IP, přičemž jednotlivé vrstvy mohou běžet na různých místech v síti.

Vrstva uživatelského rozhranní – přijímá požadavky od vývojáře a zobrazuje informace o průběhu prováděných operací. Je k dispozici buď v podobě grafického uživatelského rozhranní (Codasip Studio) nebo ve verzi příkazového řádku (CLI), která je vhodnější spíše pro pokročilé uživatele.

Middleware – úkolem middlewaru je přijímat a zpracovávat požadavky z vrstvy uživatelského rozhraní. Některé z příkazů je schopen vykonat sám (například generování nástrojů nebo funkčního verifikačního prostředí), ostatní příkazy jsou předány ke zpracování simulační vrstvě. Je také zodpovědný za instalaci simulátoru do simulační vrstvy.

Vrstva simulační – může obsahovat i více simulačních platforem, přičemž každá reprezentuje jeden procesor. Meziprocesorová komunikace je zajištěna skrze sdílené zdroje.

Na obrázku 2.1 lze vidět podrobnější hierarchii prostředí.



Obrázek 2.1: Vrstvy Codasip Studia, převzato z [2]

2.3 Codasip SDK

V této sekci si popíšeme některé dostupné nástroje (Software development kit – SDK) z prostředí Codasip Studia [2]. U každého z těchto nástrojů je možné nastavit úroveň optimalizace a určit množinu zpráv, které se budou zobrazovat na výstupu.

Assembler – tento nástroj překládá kód napsaný v jazyce symbolických instrukcí do kódu objektového. Vzniklé soubory jsou sestaveny pomocí Codasip Linkeru. Codasip Assembler dokáže zpracovat dva jazyky najednou. Jedná se o jazyk definovaný syntaxí instrukcí

z instrukční sady procesoru, druhým je jazyk, který se používá ke specifikaci direktiv a symbolů. Vygenerovaný Assembler podporuje překlad programů ve formě, která se běžně používá u GNU assembleru.

Linker – sestavuje samostatně přeložené moduly a knihovny do funkčního celku – xexe souboru. Codasip linker není nutné generovat, jelikož je nezávislý na cílové architektuře. Jedná se o rozšířenou verzi linkeru ze sady GNU-Binutils.

C/C++ překladač – Codasip Studio poskytuje přenositelný C/C++ překladač založený na LLVM platformě. Tento překladač dále rozšiřuje o podporu VLIW architektur. To umožňuje dosažení vysokého instrukčního paralelismu a vyššího výkonu s nižší frekvencí. K dispozici je i standardní knihovna C (Newlib) a knihovna Runtime. C/C++ překladač převádí aplikaci z jazyka C/C++ do GNU assembleru. Nástroj assembler poté přeloží tento program do objektového souboru ve formátu ELF. Pomocí linkeru je následně z objektových souborů vytvořen spustitelný soubor.

Simulátor – všechny typy simulátorů jsou generovány z popisu v jazyce CodAL. Simulace může probíhat buď vzdáleně – přes Codasip Middleware nebo lokálně, pokud máme vygenerované nástroje. Je možné vygenerovat 3 typy simulátorů. Prvním je simulátor zaměřující se na konstatní načítání, dekódování a provádění instrukcí (ia simulátor). Další typ se zaměřuje na hardwarovou stránku a přesné časování procesoru (ca simulátor). Posledním typem je QEMU simulátor. Jeho funkce jsou ovšem omezené – je podporována jednoprocesorová simulace a profilovací statistiky.

Debugger – Codasip debugger je součástí simulátoru. Umožňuje ladění běžících aplikací na několika úrovních a je tedy velmi důležitým nástrojem pro vývojáře softwaru. Podporuje nastavení breakpointů, watchpointů, ovládání běhu včetně krokování, zobrazování aktivovaných podprogramů či hodnot zdrojů. Lze ho také použít pro ladění na úrovni instrukcí v assembleru.

Profiler – poskytuje podrobnější informace o průběhu simulace (např. kolik hodinových cyklů trvá konkrétní funkce, které instrukce byly nejčastěji použity atd.). Získané výsledky zobrazuje ve formě seznamu s instrukcemi a funkcemi nebo v podobě grafů. Tyto informace jsou poté využívány k optimalizaci.

Kapitola 3

Paralelní a distribuované zpracování

Vývoj počítačových procesorů dosáhl v minulosti určitých omezení z hlediska neustálé potřeby zvyšovat svou rychlost. Jednalo se zejména o fyzikální limity procesorů. S postupným rozvojem mikroprocesorové technologie však bylo možné získat procesor s větším výkonem, než měly velké a drahé sálové počítače ze začátku 80.let. Tato situace vedla k rozvoji architektur založených na propojení většího množství levných procesorů do jednoho systému a tím i k vývoji paralelismu.

Paralelní a distribuované zpracování jsou způsoby výpočtu, při kterých je současně použito více výpočetních zdrojů. Řešený algoritmus je tak možné rozdělit na menší samostatné části, jež mohou být zpracovávány souběžně. Tím dosáhneme kratší doby potřebné na vykonání úlohy a zvýšení výkonu celého systému.

Tyto systémy jsou dnes hojně využívány v mnoha odvětvích, například v oblasti umělé inteligence, modelování a simulace, přírodních věd, zpracování signálů, zdravotnictví, automatizace v inženýrství apod.

3.1 Paralelní a distribuované výpočty

Ačkoliv jsou paralelní a distribuované výpočty často vzájemně zaměnovány, jedná se o dva různé přístupy. Paralelní výpočty lze chápat jako určitou pevně vázanou formu výpočtů. Všechny procesory mohou mít přístup ke sdílené paměti k výměně informací mezi procesory. Distribuované zpracování se vztahuje k širší skupině systému, zahrnující ty, které jsou pevně spojené.

Architektura paralelního výpočteního systému se dá často charakterizovat homogenitou jeho složek. Za paralelní systémy je možné považovat nejen architektury, které obsahují sdílenou fyzickou paměť a jsou označeny za jeden počítač, ale všechny architektury založené na konceptu sdílené paměti, ať už fyzicky přítomné nebo vytvořené pomocí knihoven, hardwaru a výkonné síťové architektury [15].

U distribuovaného systému probíhá distribuované zpracování. Na takovém zpracování se podílí několik nezávislých decentralizovaných jednotek, které vypadají z pohledu uživatele jako jediný systém. Hardwarové a softwarové prvky takových systémů mohou bývat heterogenní. Za distribuovaný systém je možné považovat jak systémy sestávající z více počítačů, tak i procesory/jádra na jediném počítači [4].

3.2 Klasifikace paralelních systémů

Paralelní systémy je možné klasifikovat do třech kategorií. Jedná se o multiprocesorové systémy, multipočítače a vektorové procesory [15].

Multiprocesorový systém

Jedná se o systém, ve kterém má více procesorů umožněn přímý přístup ke sdílené paměti. Ta tvoří společný adresový prostor. Taková realizace je možná použitím jediné paměti nebo kolekce paměťových modulů, které jsou úzce spojené a adresovatelné jako jedna jednotka. Tento systém obvykle odpovídá architektuře UMA (uniform memory access), pro kterou je charakteristický konstantní přístup všech procesorů ke sdíleným datům.

Procesory tohoto typu systému jsou obvykle stejného druhu a používají shodný operační systém. S pamětí jsou spojeny přes sběrnici nebo přepínače. Komunikace mezi nimi probíhá tradičně přes operace čtení a zápis na sdílenou paměť.

Multipočítačový systém

Jedná se o výpočetní systém složený z více počítačů, které spolupracují tak, že v mnoha směrech mohou být považovány za jeden systém (většinou obsahují i homogenní hardware a sotfware). Jsou spojeny pomocí vnitřní sítě. Procesory komunikují buď skrze společný adresový prostor nebo prostřednictvím zasílání zpráv. Multipočítačové systémy, které využívají společný adresový prostor odpovídají architektuře NUMA (non-uniform memory access), kdy rychlost přístupu do sdílené paměti je pro jednotlivé procesory různá.

Tyto systémy jsou obvykle nasazeny ke zlepšení výkonosti a dostupnosti oproti jedinému počítači. Typicky jsou výhodnější v poměru cenové nákladnosti i rychlosti. Mají široké spektrum využití od malých firem s několika uzly až po nejrychlejší superpočítače na světě.

Vektorový procesor

Patří do skupiny paralelních počítačů, které jsou umístěny na fyzicky společném místě, nemusí však využívat sdílenou pamět. Tato architektura byla navržena zejména pro zpracování matematických operací, zejména polí čísel. Obsahuje řadu procesorů, které pracují současně. Každý procesor používá určitý prvek pole, a tak se jediná vykonaná operace může vztahovat na všechny existující prvky paralelně.

3.2.1 Flynnova klasifikace

Z dosud publikovaných klasifikací systémů je neznámější klasifikace Flynnova z roku 1996. Ta rozděluje jednotlivé systémy z hlediska toků instrukcí a dat [8]. Schopnosti těchto kategorií jsou zobrazeny na obrázku 3.1.

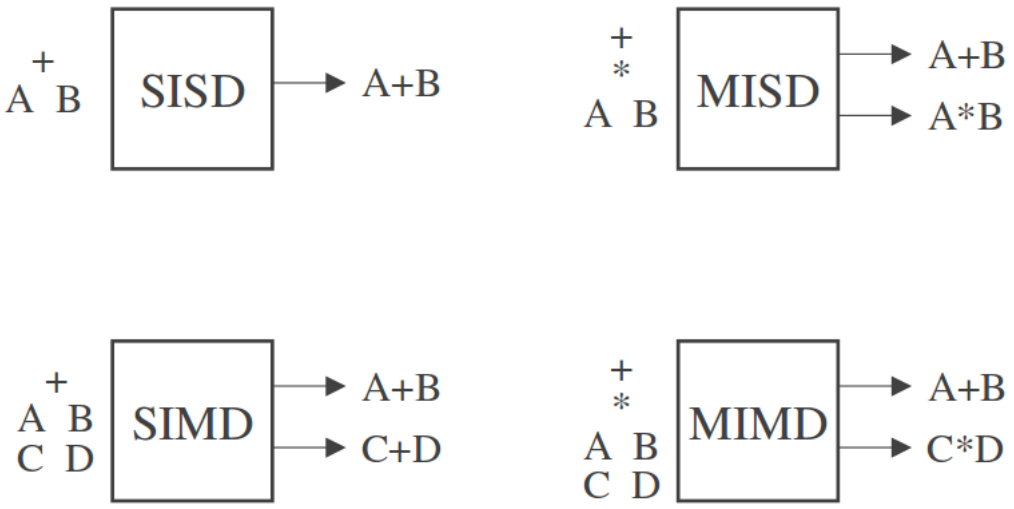
SISD (Single instruction, single data stream) – tento typ odpovídá běžnému zpracování u von Neumannovy architektury s jedním procesorem a jednou paměťovou jednotkou připojenou přes systémovou sběrnici. Data jsou zpracovávána sériově a v jednom hodinovém taktu je zpracována pouze jedna instrukce. Jedná se o nejstarší typ systémů. Paralelní zpracování je možné zajistit prostřednictvím použití několika funkčních jednotek nebo tzv. pipeliningem.

SIMD (Single instruction, multiple data stream) – u tohoto typu všechny procesory spouští stejnou instrukci, ovšem nad různými daty. Každý procesor má svá vlastní data

v lokální paměti. Vhodné uplatnění nalezneme u aplikací, které zahrnují operace nad velkými poli a maticemi, jako například vědecké programy, simulace, oblast grafiky či komprese videa, kde mohou být data snadno rozdělena. Pro některé programy může tento typ poskytnout vysoký výkon z hlediska doby provední určité operace. Oproti typu MIMD disponuje menšími paměťovými nároky, rychlejší komunikací mezi procesory a také se zde nevyskytuje potřeba složité synchroznizace mezi procesy. Nevýhodou jsou naopak aplikace, které neumožňují paralelizaci výpočtů. V důsledku toho je poté velká část systému nevyužita.

MISD (Multiple instruction, multiple data stream) – odpovídá spuštění různých operací paralelně nad stejnými daty. Procesory jsou lineárně propojené, přičemž data procházejí postupně jednotlivými procesory. Využívá se v oblasti vizualizace, krypografie, neuronových sítí či řešení úloh s proudovým charakterem. Dále se používá pro redukci možných selhání a chyb v systémech, u kterých je kladen dúraz na přesnost výsledků.

MIMD (Multiple instruction, multiple data stream) – je multiprocesorový systém, ve kterém každý procesor provádí odlišný kód nad odlišnými instrukcemi [9]. Jejich provedení může být synchronní či asynchronní, deterministické nebo nedeterministické. Jedná se o způsob provozu v distribuovaných systémech a také ve většině paralelních systémů. Do této kategorie spadají multipočítače a (vícevláknové) multiprocesory. MIMD zvyšuje možnost uváznutí (deadlocku) a kolize přístupu ke zdrojům. K eliminaci těchto rizik může být vyžadována implementace konstrukcí v podobě semaforů apod.



Obrázek 3.1: Schopnosti kategorií, převzato z [9]

3.2.2 Úrovně paralelismu

Existují tři základní úrovně, dle kterých je implementován paralelismus [18]. Jedná se o paralelismus na úrovni instrukcí, procesů/vláken a datový paralelismus. Specifikujeme si zde tyto úrovně a pro ně charakteristické architektury.

Paralelismus na úrovni instrukcí (ILP) – více instrukcí ze stejného instrukčního toku může běžet souběžně. Je generován a řízen hardwarem nebo kompilátorem. Patří sem superskalární architektury a instrukční pipelining [8].

Paralelismus na úrovni procesů/vláken (TLP) – je generován překladačem či uživatelem a řízen kompilátorem nebo hardwarem. Jedná se o typ, ve kterém může běžet více vláken či sekvencí instrukcí ze stejné aplikace zároveň. Takový paralelismus je charakteristický pro vícevláknové procesory, distribuované multiprocesory a další [8].

Datový paralelismus (DLP) – na této úrovni je jedna instrukce opakovaně použita na více datových instancí (například smyčky). Do této kategorie spadají SIMD systémy a vektorové procesory [8].

3.3 Klasifikace distribuovaných systémů

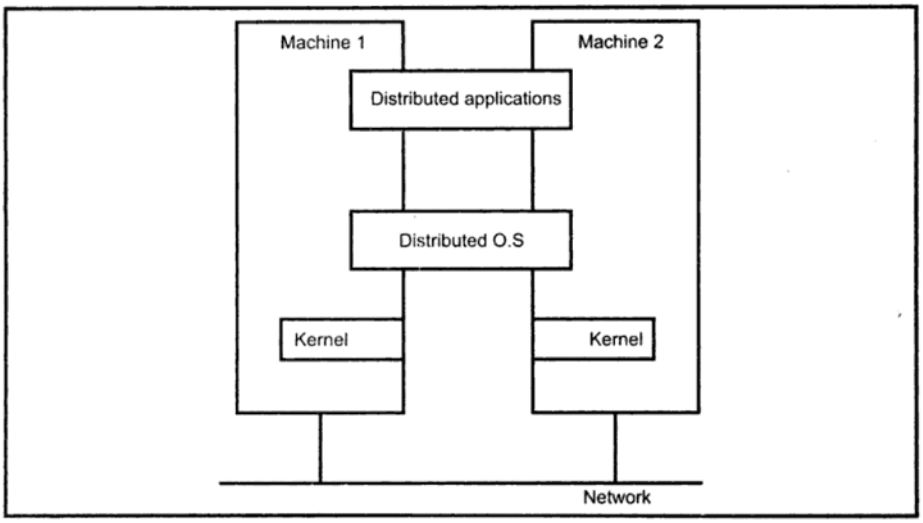
Distribuované systémy se rozdělují do dvou typů. Jsou to o multiprocesory a multipočítače. Jednotlivé typy si nyní popíšeme.

Multiprocesory

Podporují více procesorů transparentních pro aplikaci a disponují sdílenou pamětí. Je tak nutné využít synchronizačních mechanismů pro výlučný přístup ke sdíleným prostředkům. Jedná se zejména o semafory a monitory [12]. Procesory mohou mít navíc také vlastní privátní pamět. U symetrických multiprocesorů jsou všechny procesory rovnocenné (jak po hardwarové tak softwarové stránce). Každý procesor může spouštět souběžný běh několika vláken [3].

Multipočítače

Multipočítače nepoužívají sdílenou pamět. Komunikace probíhá pomocí zasílání zpráv, přičemž existuje mnoho alternativ jak zprávy předávat, například prosté HTTP, konektory RPC nebo fronty zpráv. Organizace multipočítače je znázorněna na obrázku 3.2. Jádra jednotlivých uzlů se starají o správu lokálních zdrojů, jako jsou procesory, paměť apod. Uzly disponují také vlastními moduly pro meziprocesorovou komunikaci [12].



Obrázek 3.2: Multipočítač, převzato z [4]

3.3.1 Výhody distribuovaného systému

Nyní si uvedeme několik výhodných vlastností, které poskytuje distribuovaný systém [17].

- **Sdílení prostředků** – distribuovaný systém umožňuje sdílení hardwarových či softwarových prostředků – souborů, kompilátorů, disků apod.
- **Souběžnost** – v systému může běžet několik procesů paralelně, ty mezi sebou mohou v případě potřeby komunikovat
- **Otevřenost** – tento typ systémů bývá navrhnut dle standartních protokolů, díky kterým je možné kombinovat hardware i software od odlišných dodavatelů
- **Rozšiřitelnost** – je možné navýšit výkon distribuovaného systému přidáním několika dalších komponent (procesorů, terminálů, disků apod.)
- **Tolerance k vadám** – pokud je práce rozložena na několik počítačů, tak je při výpadku některého z nich práce přerušena jen na jediném počítači. V porovnání s centralizovanými počítači by takovýto výpadek mohl způsobit přerušení běhu celého systému. U distribuovaného systému je také díky umožnění replikace dat na více počítačů možné připouštět určitá selhání softwaru nebo hardwaru. Poskytuje totiž snažší detekci a obnovu při selhání.
- **Transparentnost** – prezentování se navenek jako jeden počítačový systém. Je založen na vytvoření abstrakcí prostředků v distribuovaném systému.

Z hlediska sestavování balíků v prostředí Jenkins jsou využívány k souběžnému zpracování právě distribuované systémy, konkrétně se jedná o multipočítače. Distribuované sestavení v prostředí Jenkins je blíže popsáno v samostatné kapitole číslo 4.2.

Kapitola 4

Prostředí Jenkins

Vývoj projektu Jenkins byl pod tehdejším názvem Hudson zahájen v roce 2009 vývojářem Kohsuke Kawaguchim, pracujícím pro firmou Sun Microsystems. Postupem času se projekt rozrostl tak, že zaujímal asi 70 % podílu na trhu v oblasti softwaru týkajícího se průběžné integrace. Roku 2009 koupila firmu Sun společnost Oracle. Tento krok ale vedl postupem času ke sporům mezi původními vývojáři a vývojáři z Oracle. Ti měli zájem na pomalejším a přísně kontrolovaném procesu vývoje, kdežto Hudson vývojáři spolu s Kohsukem chtěli zachovat koncept původní, tedy otevřenost, flexibilitu a rychlý vývoj. Roku 2011 byl projekt Hudson přejmenován na Jenkins a odtržen se záměrem vrátit se k původnímu návrhu. Většina uživatelů přešla k tomuto prostředí a Oracle nadále pokračoval s vývojem odlišným směrem [7].

Jenkins je software s otevřeným zdrojovým kódem určený k zajišťování průběžné integrace při vývoji softwaru. Tento nástroj je implementovaný v jazyce Java. Disponuje jednoduchým rozhranním a dá se snadno přizpůsobit vlastním potřebám díky rozsáhlému množství zásuvných modulů, které je možné využívat. Jedná se například o moduly pro správu zdrojových kódů, spouštěče behů, notifikační nástroje, moduly pro sledování metrik a jejich vizualizaci, ukládání artefaktů, integraci s externími systémy a další.

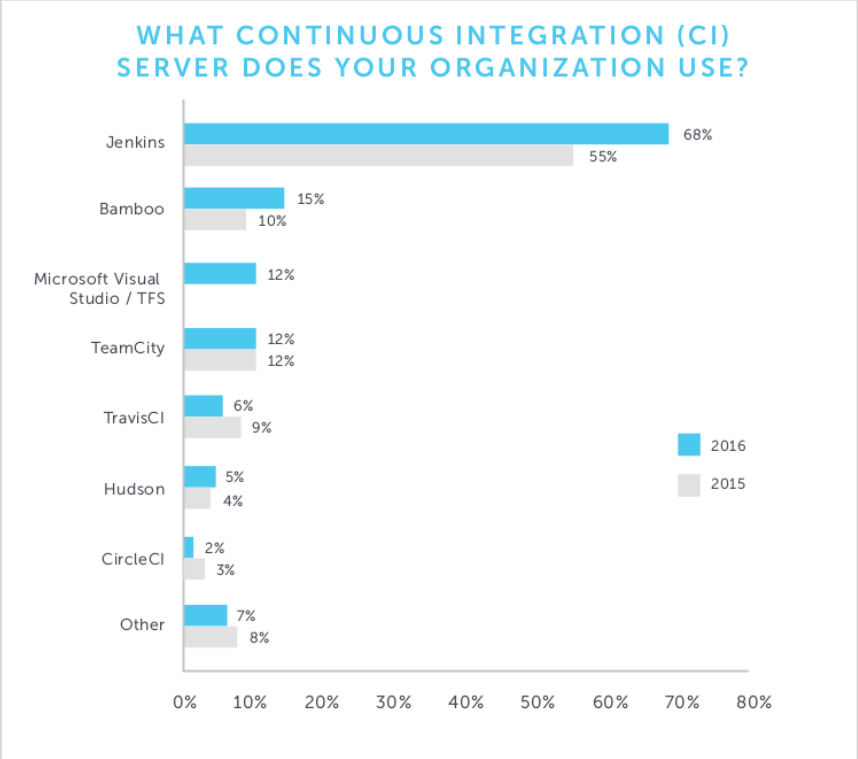
Výhodou Jenkins nástroje je jeho rychlé vývojové tempo, které neustále poskytuje nové funkce, aktualizace pluginů či opravy chyb. K dispozici je ale také stabilní verze s pomalejšími změnami a s dlouhodobou podporou. Tyto verze vycházejí jedenkrát za 3 měsíce spolu s významnými opravami chyb [7].

Dle průzkumu z roku 2016, prováděným společností Sauce Labs[1], bylo při dotazování několika set profesionálních vývojářů zjištěno, že je Jenkins nejpoužívanějším nástrojem v oblasti průběžné integrace. Konkrétní hodnoty a porovnání s dalšími existujícími nástroji jsou uvedeny v grafu číslo 4.1. Celý článek je k dispozici zde [16].

4.1 Průběžná integrace

Před nástupem průběžné integrace byla tvorba softwaru spojena s několika nevýhodami. Vývojářské týmy prováděli průběžné spojování kódu a změn v nich nesystematickými metodami a bylo mnohdy nutné některé části kódu znovu přepisovat. Takový proces mohl trvat i několik týdnů či měsíců. Vzniklé problémy mnohdy vedly k neplánované práci navíc a nevyhnutelnému zpoždění při vydávání.

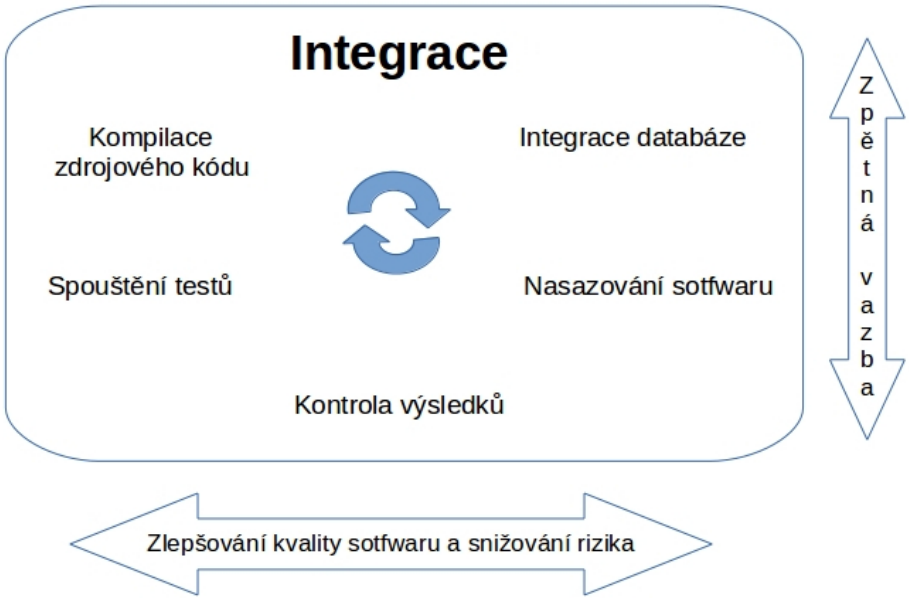
[1] https://saucelabs.com/



Obrázek 4.1: Umístění serveru Jenkins v žebříčku nejvyužívanějších nástrojů pro průběžnou integraci, převzato z [16]

Pro moderní vývoj softwaru je dnes velice výhodné použití prostředků, které urychlují a zjednodušují jeho tvorbu. Tvorba softwaru je v dnešní době proces, zahrnující psaní samotného kódu, ale také zejména sledování jeho změn, průběžné testování, kontrolu kvality kódu, tvorbu metrik a v neposlední řadě nasazení výsledného produktu. Průběžná integrace (Continuous integration – CI) umožňuje zjednodušit takovýto proces vývoje urychlením a automatizací některých částí. Pomáhá k snadnějšímu odhalování chyb a jejich reportování, rychlou detekci změn v kódu, automatickému testování nových verzí či automatizaci procesu nasazení.

Využívání průběžné integrace ovšem vyžaduje přizpůsobení takovému vývoji (viz obrázek 4.2). Proces sestavení projektů by měl být spolehlivý, opakovatelný a měl by být vytvořen tak, aby u něj nebylo nutné uživatelského zásahu. Při ukládání změn a jejich přenosu do verzovaného repozitáře by si měl být uživatel jistý, že je kód funkční. Také i oprava neúspěšného buildu by měla být nepřehlédnutelnou součástí práce na projektu a neměla by být oddalována. Kvalitní vývoj se samozřejmě neobejde bez tvorby testů a jejich pravidelného spouštění. Testování před samotným nasazením softwaru dokáže odhalit mnoho chyb a proto je vždy nutné se zaměřit na vhodné testovací postupy a klást důraz na vysokou kvalitu testů [6].



Obrázek 4.2: Vizualizace integrace

4.2 Architektura distribuovaného sestavení

Jenkins umožňuje vykonávat činnosti ve dvou režimech. V běžném režimu je využíván pouze jeden stroj (uzel), který pracuje samostatně. Druhým režimem je architektura Master/Slave. Jelikož je tato architektura využívána v aktuálním procesu sestavení balíku Codasip Studio, uvedeme si nyní blíže její princip.

Master/Slave architektura

Architektura Master/Slave je založena na zapojení více uzlů do jednoho projektu. Tento způsob dovoluje distribuovat potřebnou práci z jednoho stroje do několika menších jednotek. Kromě snížení vytížení jediného uzlu se tento princip také využívá při potřebě existence rozličných prostředí, za účelem testování, vývoje pro více platforem apod [7].

Hlavním uzlem projektu je uzel master, na kterém je spuštěn Jenkins server. Jeho úlohou je plánovat a řídit běh všech podřízených slave uzlů a zpracovávat jejich výsledky. Úkolem slave uzlů je vykonávat práci, zahrnující spouštění běhů expandovaných masterem.

Slave je spustitelný Java soubor, běžící na vzdáleném počítači. Uzly tohoto typu mohou běžet na různých operačních systémech. Komunikace s hlavním uzlem probíhá pomocí protokolu TCP/IP. Je možné nakonfigurovat projekt tak, aby vždy běžel na konkrétním slave uzlu. Další možností je definovat skupinu uzlů, ze které se uzel vybírá nebo lze nechat v režii prostředí Jenkins zvolit libovolný volný stroj.

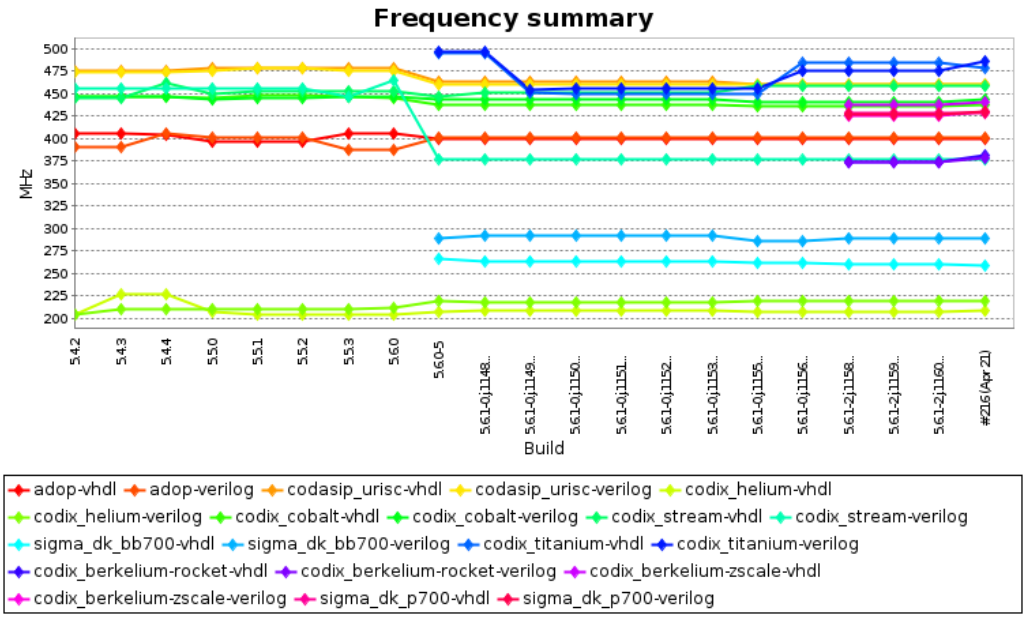
Jenkins umožňuje kromě paralelního spouštění na více uzlech také spuštění souběžných běhů na jediném uzlu. Každý z běhů přitom disponuje vlastním pracovním adresářem. Počet možných běhů určují prostředky, zvané exekutory. Pokud není nastaveno jinak, má uzel k dispozici jeden exekutor, který může být přiřazen určité úloze, jež se má vykonat. Tento typ paralelismu však není vhodný pro sestavování balíku Codasip Studio, jelikož je tento proces náročný na výpočetní výkon stroje a je proto vhodnější využít více uzlů, které nebudou omezeny sdílením výpočetních prostředků mezi jednotlivými úlohami.

4.3 Využití pro Codasip Studio

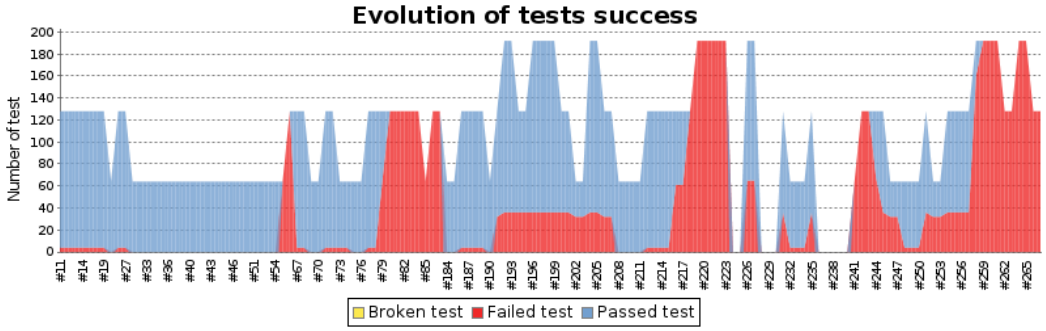
Prostředí Jenkins je využíváno v celém integračním procesu, zahrnuje periodické spouštění buildů, parametrizaci běhu, podmíněné spouštění navazujících projektů v závislosti na úspěšném dokončení, spouštění shell skriptů, zajištění různých konfigurací projektů, spolupráci s verzovacím systémem Git nebo notifikaci o stavu projektu pomocí zasílání e-mailů.

Dále je v tomto prostředí prováděno pravidelné noční testování. Jedná se například o testování nástrojů jako je assembler, debugger, překladač, profiler, testování funkčního verifikačního prostředí a funkčních jednotek, ověřování správného generování random aplikací, testování grafického vývojového prostředí Codasip IDE, generování hardwaru, testování IP komponent, instalátoru a dalších částí produktů Codasip.

Důležité je také uchovávání výsledků jednotlivých běhů nebo například vizualizace získaných hodnot, jež může poskytnout přehled z hlediska dlouhodobého vývoje apod. Ukázka vizualizace výsledků v podobě grafu je znázorněna na obrázku 4.3. Užitečnou funkcí je také sledování poměru úspěšnosti testů (viz obrázek 4.4). Samozřejmá je i automatizace umístění vytvořených balíků na server a mnoho dalších zajímavých funkcí.



Obrázek 4.3: Vizualizace hodnot



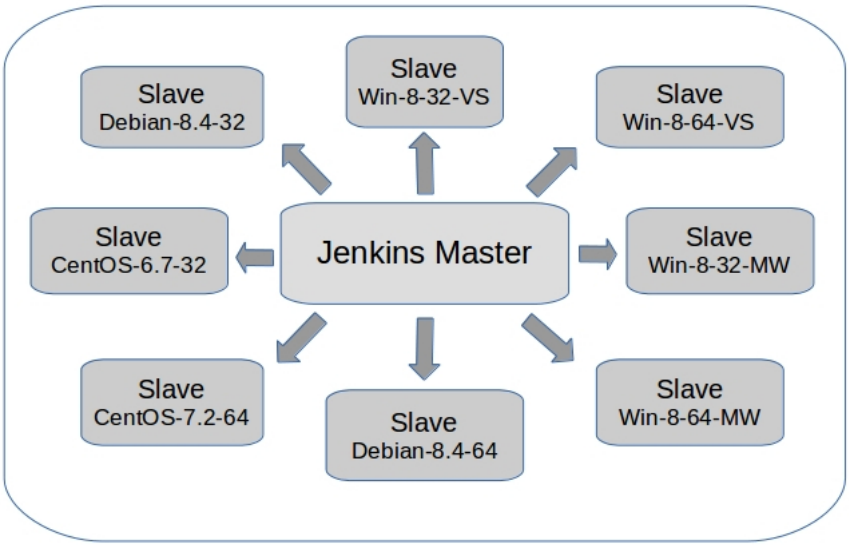
Obrázek 4.4: Úspěšnost testů

Kapitola 5

Aktuální průběh sestavování balíku

Vytváření balíku vývojového prostředí Codasip Studio firmou Codasip je automatizovanou činností. Balíky se vytváří každý den a při úspěšném sestavení probíhá jejich nahrání na server, kde jsou k dispozici uživatelům. Pro tvorbu těchto balíků v prostředí Jenkins je nyní využíván pouze jediný samostatný projekt. Jenkins poskytuje vytvoření několika typů projektů. Jedná se o Freestyle software project, Maven project, Build Flow, External job a Multi-conguration project [7]. Z nich je k sestavení balíku využíván právě poslední typ – multikonfigurační projekt. Tento typ umožňuje spouštění stejného procesu sestavení na více strojích s různými konfiguracemi. V takové sestavě je vždy přítomen jeden stroj, který ovládá činnost strojů ostatních. Vytváří se tak distribuovaný běh (viz kapitola 4.2), ve kterém je stroj master hlavním uzlem, jež řídí činnost podřízených slave uzlů a deleguje na ně potřebné úlohy.

Jelikož je nutné vytvářet balíky pro několik typů operačních systémů (CentOS, Debian, Windows), pro 32bitové i 64bitové architektury a u systémů Windows navíc pro typy Visual Studio a Mingw, využívá se tento typ projektu k zahájení tvorby balíků na všech systémech paralelně. Přehled všech konfigurací je znázorněn na obrázku 5.1.

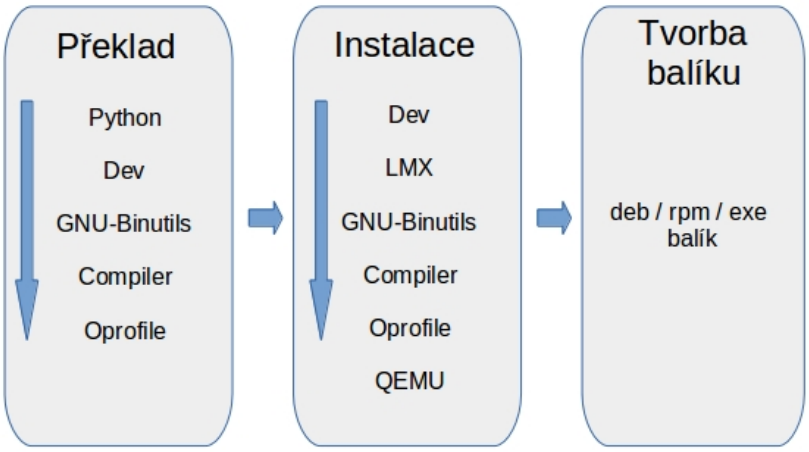


Obrázek 5.1: Tvorba balíků Codasip Studio – master/slave architektura

Konvence pojmenování uzlů je následující: `systém-verze-32/64_bitová_architektura`. Pro systém Windows je navíc za poslední pomlčkou rozlišena verze balíku – `VS` pro Visual Studio a `MW` pro typ Mingw.

Aktuální vytváření balíku je rozděleno do 3 hlavních fází. Jedná se o samotný překlad zdrojových kódů, instalaci do zvoleného adresáře a vytvoření konečné podoby balíku (typu deb pro systémy Debian, rpm pro systémy CentOS či instalačního balíku exe pro platformu Windows) v závislosti na sestavující architektuře.

Překlad zdrojových kódů je nyní prováděn sériově v tomto pořadí: Python, Dev, GNU-Binutils, Compiler a OProfile. Instalace do cílového adresáře probíhá v pořadí Dev, LMX, GNU-Binutils, Compiler, OProfile a QEMU. Celý postup je zobrazen na obrázku 5.2.



Obrázek 5.2: Sériové sestavení balíku

Při tvorbě návrhu se budeme držet posloupnosti jmenovaných hlavních fází, avšak jednotlivé kroky překladu a instalace se pokusíme paralelizovat.

Kapitola 6

Návrh paralelizace sestavení

V této kapitole si popíšeme vzájemné vztahy kompilovaných a instalovaných částí, zaměříme se na délku jejich překladu a možnosti paralelizace a poté navrhneme způsoby, jakými můžeme v prostředí Jenkins realizovat potřebné požadavky.

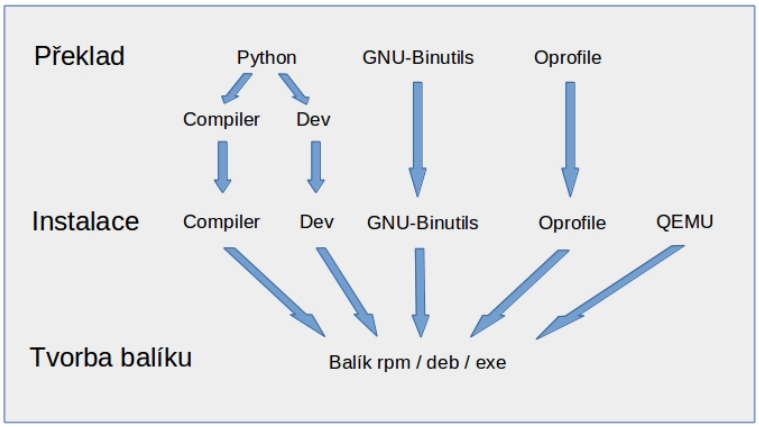
6.1 Rozdělení běhu

Ze všeho nejdříve je nutné zjistit závislosti mezi jednotlivými částmi, abychom mohli určit potřebnou posloupnost překladu, instalace a také to, které z těchto úloh mohou běžet souběžně.

- Python – intepret jazyka Python. Je nutné aby byl přeložen před sestavením částí Compiler a Dev, jelikož je pro jejich překlad potřebná jeho aktuální verze. Může se kompilovat samostatně. Do výsledného balíku se jeho kód nevkládá.
- GNU-Binutils – jedná se o modifikovanou kolekci programovacích nástrojů GNU-Binutils, navíc podporující rozšířený objektový formát firmy Codasip. Je možné překládat samostatně bez potřeby ostatních částí.
- Dev – obsahuje příkazový řádek, assembler, diassembler, simulátory, kompletni popis struktury jazyka CodAL a další. Pro svůj překlad vyžaduje zkompilovanou část Python.
- Compiler – jedná se o část backend prekladače, clang. Vyžaduje zkompilovaný Python.
- OProfile – modifikovaná verze statického profilovacího nástroje OProfile. Je možné jej překládat samostatně bez přítomnosti ostatních částí.
- QEMU – emulační platforma, kterou využívá firma Codasip pro simulaci. Přeložitelná část je obsažena v části Dev. Druhá část je umístěna v samostatném repozitáři. Tato část se nepřekládá, probíhá pouze její instalace, jež je nezávislá na ostatních částech.

Na obrázku 6.1 můžeme vidět potřebnou posloupnost při překladu. Následná instalace částí Dev, LMX, GNU-Binutils, Compiler, OProfile a QEMU může být prováděna nezávisle v jakémkoli pořadí. Pro tvorbu balíku je nutná přítomnost všech instalovaných částí.

Z popisu na uvedém obrázku si také můžeme všimnout, že paralelizace bude možná hned v několika místech. Zaměříme se tedy na délku samotných úseků, abychom mohli rozložit co nejlépe práci na jednotlivých strojích.



Obrázek 6.1: Závislosti při překladu a instalaci

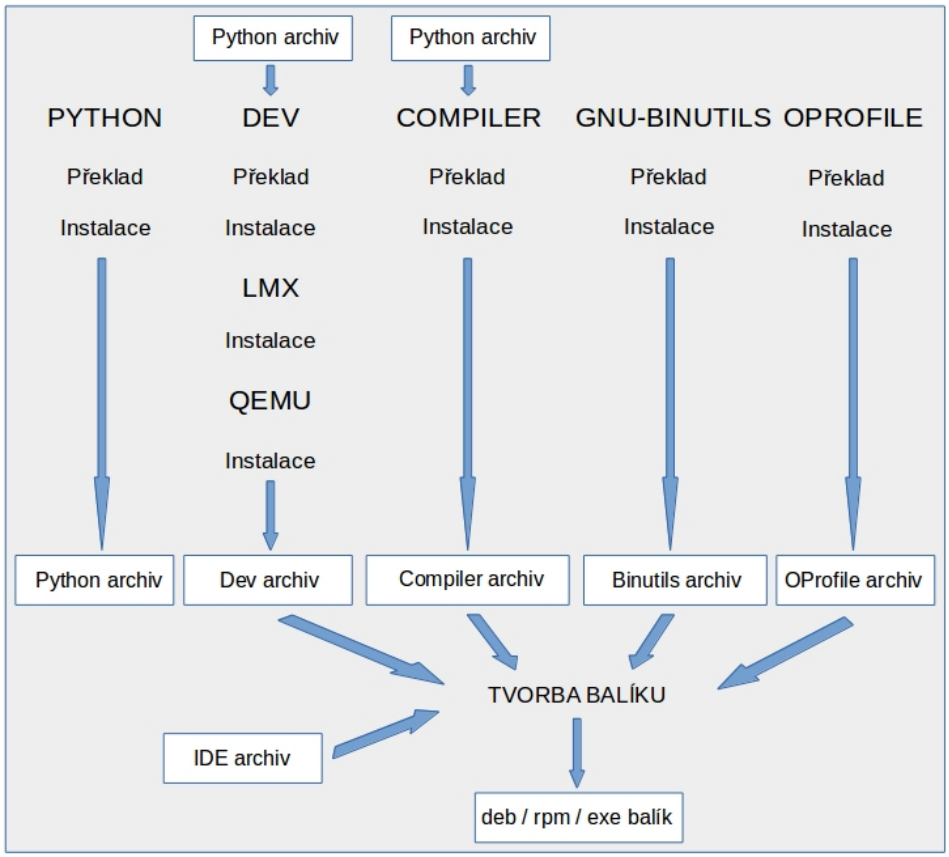
Z aktuálních konfigurací využívaných k tvorbě balíku (viz kapitola 5) byly vybrány dvě konfigurace, na kterých bude prováděno testování doby běhu jednotlivých částí a následná implementace paralelizace. Jedná se o 64bitové systémy CentOS 6.7 a Windows 8[1].

	Část	**Uzel**	**Doba** (h:m:s)
Překlad	Python	CentOS-6.7-64	00:04:40
		Win-8-64-VS	00:00:01
	Dev	CentOS-6.7-64	00:40:51
		Win-8-64-VS	00:24:09
	GNU-Binutils	CentOS-6.7-64	00:04:24
		Win-8-64-VS	00:15:11
	Compiler	CentOS-6.7-64	01:19:07
		Win-8-64-VS	00:47:46
	OProfile	CentOS-6.7-64	00:02:36
		Win-8-64-VS	00:00:02
Instalace	Dev	CentOS-6.7-64	00:00:11
		Win-8-VS-64	00:00:36
	LMX	CentOS-6.7-64	00:00:01
		Win-8-VS-64	00:00:01
	GNU-Binutils	CentOS-6.7-64	00:00:01
		Win-8-64-VS	00:00:05
	Compiler	CentOS-6.7-64	00:00:38
		Win-8-64-VS	00:04:01
	OProfile	CentOS-6.7-64	00:00:01
		Win-8-64-VS	00:00:01
	QEMU	CentOS-6.7-64	00:00:08
		Win-8-64-VS	00:00:44
Tvorba balíku	rpm/exe	CentOS-6.7-64	00:14:30
		Win-8-64-VS	00:15:43

Tabulka 6.1: Délka běhu jednotlivých úseků

[1]Systémy disponují 2 procesory a 4GB paměti RAM

Z tabulky číslo 6.1 lze vidět, že nejdelší dobu běhu zaujímá překlad částí Compiler a Dev. Kompilace těchto dvou částí není vzájemně závislá a tak je možné ji provádět na dvou uzlech paralelně, aby se co nejvíce zkrátila doba překladu. Abychom ušetřili čas, který by vyžadoval překlad Pythonu, potřebného pro obě části (Compiler a Dev), vyčleníme ho do samostatného projektu. Aktuálně přeložený Python si z tohoto projektu mohou části Compiler a Dev stáhnout prostřednictvím zásuvných modulů prostředí Jenkins (viz dále). Části GNU-Binutils a OProfile je možné také umístit do samostatných projektů. Přeložené soubory je totiž nutné pouze vložit do cílového balíku a nemusí být u překladu ostatních částí přítomny. Rozdělení do projektů je znázorněno na obrázku 6.2.



Obrázek 6.2: Rozdělení částí do projektů

Spolu s instalací Devu po jeho překladu probíhá ve stejném projektu také instalace částí LMX a QEMU. Jedná se totiž o pouhé kopírování několika adresářů se soubory a není tedy nutné pro tyto části vytvářet samostatné projekty, které by dobu tvorby balíku zbytečně prodloužily. V projektech Compiler, GNU-Binutils a OProfile je také zajištěna následná instalace zkompilovaných souborů do cílového adresáře. Do výsledného balíku se také vkládá archiv s grafickým prostředím Codasip IDE. Tvorba tohoto archivu je řešena v samostatném projektu prostředí Jenkins a není předmětem paralelizace v této práci.

6.2 Paralelizace v prostředí Jenkins

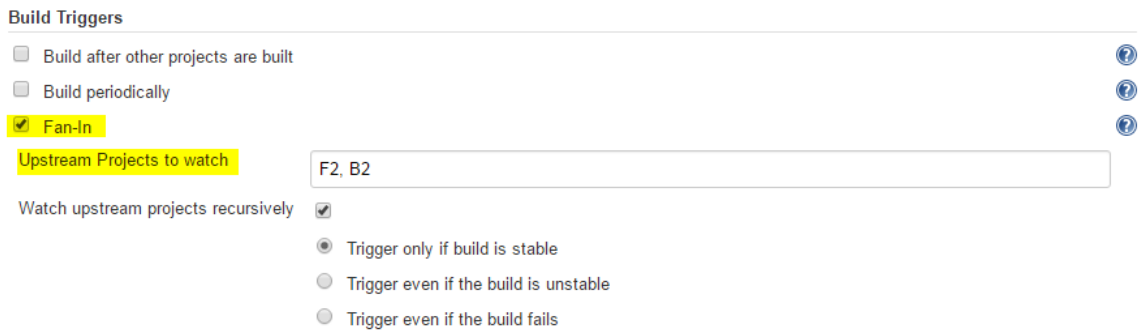
V této sekci si popíšeme, jaké možnosti nabízí prostředí Jenkins pro paralelizaci běhů, jmenujeme zásuvné moduly zajišťující potřebnou funkčnost a uvedeme si konkrétní využití těchto modulů v našem návrhu tvorby balíku.

6.2.1 Join uzly

Jelikož je sestavení balíku rozděleno do několika projektů, bude potřeba, aby před jeho konečným sestavením byly dokončeny všechny kompilace a instalace. Na tuto práci je možné v prostředí Jenkins využít tzv. Join uzly (Joins), které slouží jako místo, kam se sbíhají vybrané projekty po jejich dokončení. Při dosažení takového uzlu všemi požadovanými projekty je možné aktivovat běh dalších projektů. Takové chování využijeme i v paralelním běhu částí Compiler a Dev, kdy po dokončení obou z nich bude možné spustit finální projekt, který bude sestavovat kompletní balík. Zásuvné moduly umožňující tuto funkčnost jsou JoinPlugin a modul JobFanIn.

Zásuvný modul JobFanIn

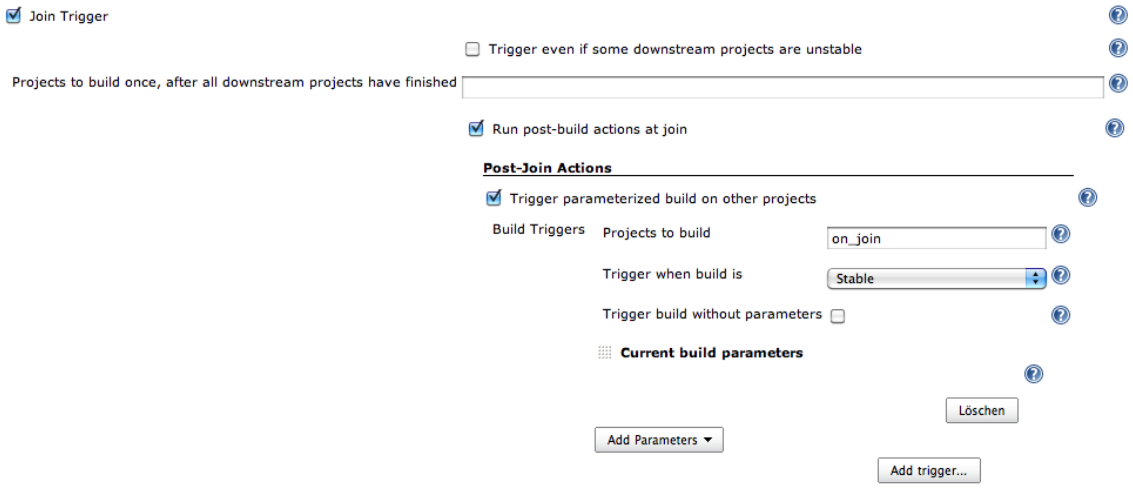
Umožňuje sledovat nadřazené projekty a po doběhnutí všech nadřazených, spustit projekty podřazené. Je možné nastavit podmínku spuštění podřazených projektů na tři různé varianty – spuštění pouze tehdy, pokud všechny nadřazené projekty proběhly úspěšně, spustit i pokud skončily nestabilní nebo spustit i v případě že jejich běh selhal (obrázek 6.3). Zásuvný modul dokáže sledovat stabilitu projektů rekurzivně – pro nadřazené projekty nadřazených projektů atd [13].



Obrázek 6.3: Zásuvný modul JobFunIn – možnosti nastavení

Join Plugin

Zásuvný modul s názvem Join Plugin [5] je užitečný pro vytvoření závislosti ve tvaru diamantu, což znamená, že jeden rodičovský projekt zahájí několik podřazených projektů paralelně. Jakmile jsou všechny dokončeny, je spuštěn finální projekt. Takovou hierarchii je vhodné využít při sestavování balíku, jelikož je tak možné spustit v rodičovském projektu překlad a instalaci potřebných úseků a poté tyto části v jediném podřazeném projektu sloučit a sestavit z nich balík. Další výhodou tohoto zásuvného modulu je možnost spuštění podřazeného projektu spolu s předáním potřebných parametrů. Je tak možné sdílet důležité údaje mezi projekty, jako je například jméno verze, revize balíku apod. Na obrázku 6.4 je zobrazena možnost konfigurace zásuvného modulu v prostředí Jenkins.



Obrázek 6.4: Join Plugin – možnosti nastavení, převzato z [5]

6.2.2 Artefakty

Po překladu či instalaci částí distribuovaných v Jenkins projektech je nutné získané soubory uložit takovým způsobem, aby byly k dispozici ostatním projektům, které je potřebují. Může se jednat jak o soubory sloužící pro navazující překlad (Python) nebo o sloučení instalačních adresářů (Compiler, GNU-Binutils, OProfile a Dev s instalací LMX a QEMU) pro tvorbu balíku.

Prostředí Jenkins nabízí možnost uložit jakékoliv soubory či adresáře z pracovní plochy projektu jako artefakty. Ty je poté možné kopírovat do jiných projektů. K tomuto kroku je potřeba doinstalovat do prostředí Jenkins zásuvný modul Copy Artifact Plugin [10].

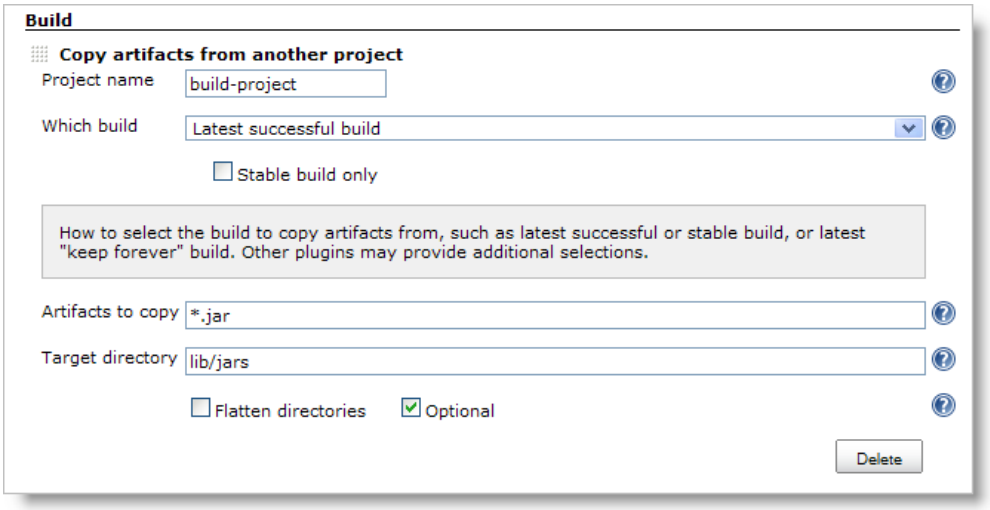
Copy Artifact Plugin

Přidá další úlohu, která se má vykonat za běhu projektu. Touto úlohou je kopírování artefaktů z jiných projektů do námi vybraného adresáře v pracovní ploše. Poskytuje možnost určit, který běh projektu zvolit – poslední úspěšný, poslední stabilní, číslo konkrétního běhu nebo výběr pomocí zadaného parametru. Ke kopírování můžeme také použít filtr k vybrání všech potřebných artefaktů apod. Na obrázku 6.5 lze vidět umožněná nastavení.

6.2.3 Dotazování

Přestože je sestavovaní balíku prováděno každý den, není nutné kompilaci částí Python, GNU-Binutils a OProfile provádět v tak krátkém intervalu. Změny v repozitářích odpovídajících těmto částem totiž nejsou tak časté, aby vyžadovali každodenní překlad. Bylo by tedy vhodné zajistit, aby se překlad provedl pouze tehdy, pokud nastala v dané části určitá změna.

Zásuvný modul Git Plugin [14] nabízí možnost dotazování se (anglicky Polling) na odlišnosti v kódu určitého repozitáře verzovacího systému od doby, kdy byla provedena poslední kontrola. Jsou-li zjištěny nějaké změny, automaticky se daný projekt spustí. Dotazování je možné nastavit na jakýkoliv interval (minuty, hodiny, měsíce apod.), který potřebujeme. Čím kratší je interval dotazování, tím dříve je možné zjistit změny v repozitářích a vytvo-



Obrázek 6.5: Copy Artifact Plugin – možnosti nastavení, převzato z [10]

řit aktuální verzi balíku. Tato operace je však nákladná, jelikož vyžaduje skenování celého pracovního prostoru a jeho ověřování se serverem.

Lepší variantou je zajistit, aby samotná změna v kódu automaticky signalizovala související projekt a ten byl následně spuštěn. K tomu lze využít tzv. Hooky [11], které poskytují oznámení o změně stavu ve verzovacím systému. Oznámení se v takovém systému nastavují k jednotlivým repozitářům. Následně je potřeba pouze umožnit prostředí Jenkins aby tyto informace detekoval. Využijeme k tomu již zmíněný Git Plugin, který dovoluje nastavit detekci potřebných Hooků v projektech.

Části Compiler a Dev tak budou moci v okamžiku jejich spuštění pouze stáhnout do své pracovní plochy zkompilovanou část Pythonu jako artefakt a tím se ještě více sníží celkový čas tvorby balíku. Stejným způsobem se budou aktualizovat artefakty pro GNU-Binutils a OProfile, které se kopírují do finálního projektu.

6.2.4 Proměnné stroje

Při práci s artefakty je nutné konkrétně identifikovat konfiguraci projektu, ze kterého se mají potřebné soubory kopírovat. Jenkins umožňuje definovat proměnné stroje (node variables), pomocí nichž je možné zajistit veškeré tyto důležité informace. Pro vytvoření proměnné stačí pouze nastavit její jméno (identifikátor) a hodnotu, která bude specifická pro konkrétní konfiguraci.

V těchto proměnných je třeba nést údaje o tom, odkud se má kopírovat část Python v závislosti na architektuře, na které probíhá překlad Compiler a Devu. Dále je také nutné upřesnit konfigurace, ze kterých se budou stahovat zkompilované a nainstalované části OProfile, GNU-Binutils, Dev a Compiler do finálního projektu.

Kapitola 7

Implementace

Implementace paralelizace vychází z návrhu vytvořeném v kapitole číslo 6. V této kapitole je popsána realizace pro vybrané typy konfigurací – jedná se o 64bitový systém CentOS 6.7 a 64bitový systém Windows 8 pro typ Visual Studio. Principy implementace u ostatních konfigurací (viz obrázek 5.1 v kapitole 5) se neodlišují od těchto dvou uvedených.

7.1 Rozdělení do projektů

Ze všeho nejdříve vytvoříme v prostředí Jenkins pět multikonfiguračních projektů. V níže uvedené tabulce 7.1 můžeme vidět jejich názvy a také části, které těmto projektům odpovídají.

Projekt	**Část**
Parallel-build-compiler	Compiler
Parallel-build-dev	Dev, LMX, QEMU
Parallel-build-binutils	GNU-Binutils
Parallel-build-oprofile	OProfile
Parallel-build-python	Python

Tabulka 7.1: Multikonfigurační projekty

U každého projektu je potřeba specifikovat, na kterých uzlech budou běžet. Budeme-li vycházet z tabulky 6.1 z kapitoly 6, dle které nejdelší dobu překladu zaujímá část Compiler, pokusíme se tuto dobu zkrátit tak, že vyhradíme pro tento projekt uzel s vyšším množstvím přiřazených procesorů. Pro 64bitový systém CentOS 6.7 tak bude překlad probíhat na stroji se čtyřmi procesory místo původních dvou. Tomuto uzlu přiřadíme i projekty Parallel-build-oprofile a Parallel-build-python, abychom co nejvíce využili potenciál stroje. Pro 64bitový systém Windows 8 je možnost využití více procesorů bohužel zbytečným krokem, jelikož firma Codasip momentálně podporuje na strojích Windows pouze jednoprocesorový překlad. Bude tedy pro zmíněné části použit původní stroj se dvěma procesory.

Část GNU-Binutils se v každém případě překládá pomocí kompilátoru GCC, neexistuje varianta pro typ Visual Studio (VS). Je tak možné pro oba typy balíků překládat tuto část na jediném stroji – typu Mingw (MW). Pro projekty Parallel-build-dev a Parallel-build-binutils využijeme stroje se třemi procesory u systému CentOS a u systému Windows vybereme dostupný uzel se dvěma procesory. Jednotlivé uzly jsou znázorněny v tabulce 7.2.

Identifikátor za poslední pomlčkou značí server či stroj, na kterém uzel běží.

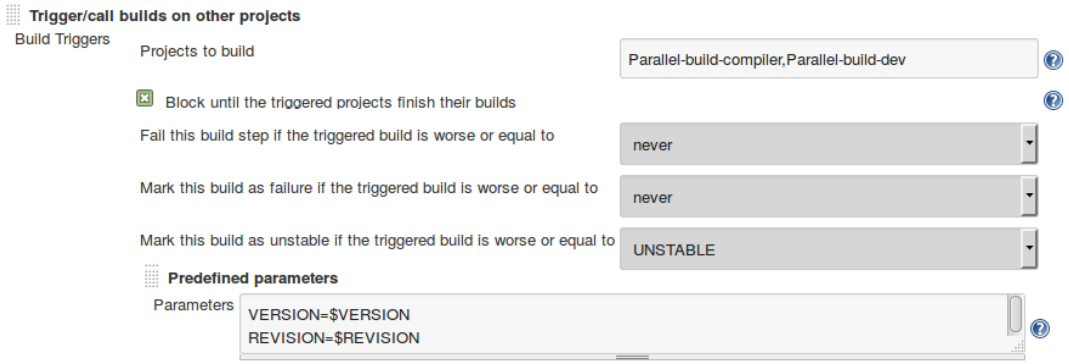
Projekt	**Uzel**
Parallel-build-compiler	CentOS-6.7-64-C5
	Win-8-64-VS-build
Parallel-build-dev	CentOS-6.7-64
	Win-8-64-VS-C4
Parallel-build-binutils	CentOS-6.7-64
	Win-8-64-MW-new
Parallel-build-python	CentOS-6.7-64-C5
	Win-8-64-VS-build
Parallel-build-oprofile	CentOS-6.7-64-C5
	Win-8-64-VS-build

Tabulka 7.2: Uzly přiřazené k projektům

Vzhledem k tomu, že se projekty Parallel-build-oprofile, Parallel-build-binutils a Parallel-build-python budou spouštět pouze v případě změny v některých z jim příslušících repozitářů, není nutné zajišťovat nutnost čekání na jejich dokončení před konečnou fázi sestavování balíku. To ale neplatí pro projekty Parallel-build-compiler a Parallel-build-dev. Tyto projekty budou vždy spouštěny zároveň a je také nutné, aby byla činnost finálního projektu zahájena až po dokončení obou z nich.

K tomuto účelu tedy vytvoříme v prostředí Jenkins hlavní projekt s názvem Parallel-build. Tento projekt již nemusí být typu multikonfigurační, ale stačí zvolit základní typ projektu – Freestyle Build Job. Na jeho práci bude totiž potřebný pouze jediný uzel. Můžeme zvolit jakýkoliv dostupný uzel v prostředí Jenkins, kromě uzlů, přiřazených k projektům Parallel-build-compiler a Parallel-build-dev. Tyto projekty totiž bude mít hlavní projekt Parallel-build za úlohu spouštět a také čekat na jejich dokončení. V případě, že by hlavní projekt běžel na uzlu, který je nastavený i v konfiguraci některého podřízeného projektu, bylo by nemožné běh na dané konfiguraci spustit.

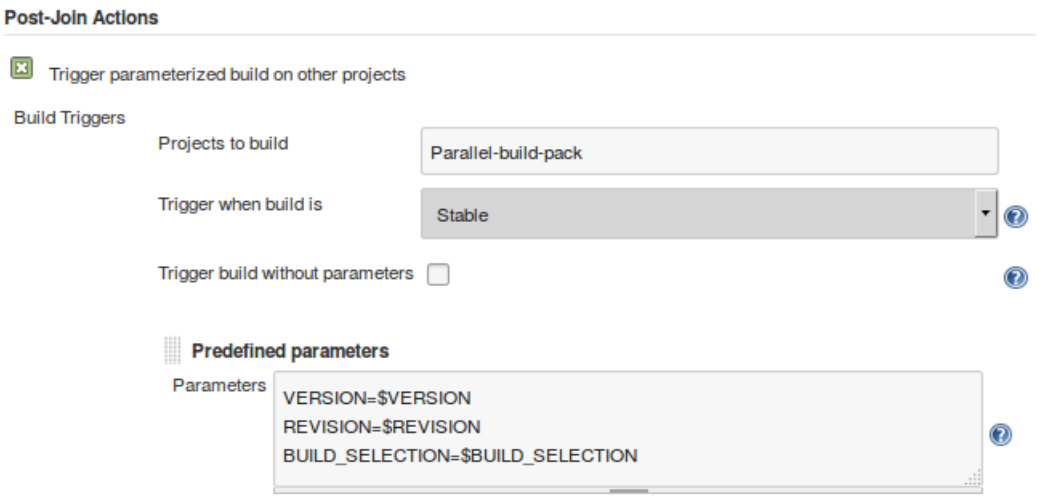
V hlavním projektu tedy nastavíme položku určení uzlu, na kterém se má běh projektu spouštět, na hodnotu Debian-8.4-64. K tomu, abychom nastavili spouštění podřazených projektů z hlavního projektu, přidáme tento krok, znázorněný na obrázku 7.1, jako další úlohu do konfigurace projektu.



Obrázek 7.1: Nastavení spouštění podřazených projektů

Nastavíme zde jména projektů, které se mají spustit, zaškrtneme položku blokování pokračování v běhu dokud projekty nebudou dokončené a dále specifikujeme proměnné, které se mají předat do podřazených projektů (hodnotu verze a revize a balíku).

Posledním projektem, který je potřeba vytvořit, je projekt Parallel-build-pack, kde bude probíhat finální tvorba balíku (rpm, deb či exe). Tento projekt bude opět multikonfiguračním, kvůli potřebě vytvářet balíky pro různé systémy. Pro 64bitový systém CentOS 6.7 a 64bitový systém Windows 8 (typ Visual Studio) vybereme uzly CentOS-6.7-64-C5 a Win-8-64-VS-build. K tomu, aby byl tento projekt spuštěn po dokončení projektů Parallel-build-compiler a Parallel-build-dev je nutné přidat tuto úlohu k hlavnímu projektu Parallel-build jako poslední krok, který má tento projekt vykonat. Potřebné nastavení je znázorněno na obrázku 7.2.



Obrázek 7.2: Nastavení spouštění finálního projektu

Vytvořené vazby, které zobrazuje Jenkins na stránce u hlavního projektu je možné vidět na obrázku 7.3.

Obrázek 7.3: Vazby mezi projekty

Stahování repozitářů

V prostředí Jenkins je nutné ke každému projektu nastavit, jaké zdrojové kódy ze kterých GitLab repozitářů se mají při spuštění běhu stahovat. Jelikož stáhnutí některých z nich trvá mnohdy i několik minut, nastavíme tedy ke každému projektu pouze ty repozitáře s konkrétní větví, které jsou potřebné pro daný překlad, instalaci nebo tvorbu balíku.

Proměnné stroje

Nyní je potřeba nastavit hodnoty proměnných ke strojům, jež provádějí stahování artefaktů z jiných projektů. Dle obrázku 6.2 z kapitoly číslo 6 lze vidět, že je nutné nastavit u projektů Parallel-build-compiler a Parallel-build-dev specifikátor, určující odkud se do jejich pracovní plochy bude kopírovat archiv s přeloženou částí Python. Dále můžeme vidět, že je třeba specifikovat konfigurace, ze kterých se budou stahovat potřebné artefakty na pracovní plochu projektu Parallel-build-pack, kde se tvoří výsledný balík. Potřebné proměnné a jejich hodnoty jsou vypsány v tabulce 7.3.

Uzel	**Název proměnné**	**Hodnota**
CentOS-6.7-64-C5	SPECIFIER_BTLS	CentOS-6.7-64
	SPECIFIER_DEV	CentOS-6.7-64
	SPECIFIER_PYTHON	CentOS-6.7-64-C5
	SPECIFIER_COMPILER	CentOS-6.7-64-C5
Win-8-64-VS-build	SPECIFIER_BTLS	Win-8-64-MW-new
	SPECIFIER_DEV	Win-8-64-VS-C4
	SPECIFIER_PYTHON	Win-8-64-VS-build
	SPECIFIER_COMPILER	Win-8-64-VS-build
CentOS-6.7-64	SPECIFIER_PYTHON	CentOS-6.7-64-C5
Win-8-64-VS-C4	SPECIFIER_PYTHON	Win-8-64-VS-build

Tabulka 7.3: Hodnoty proměnných přiřazených ke strojům

7.2 Artefakty

Pro práci s artefakty je nutné v prostředí Jenkins nastavit několik informací. Jedná se jak o způsob ukládání samotného artefaktu, povolení jeho stažení určitým projektem a také specifikaci určující ze kterého běhu a kterého projektu se má artefakt stahovat.

Ukládání artefaktů

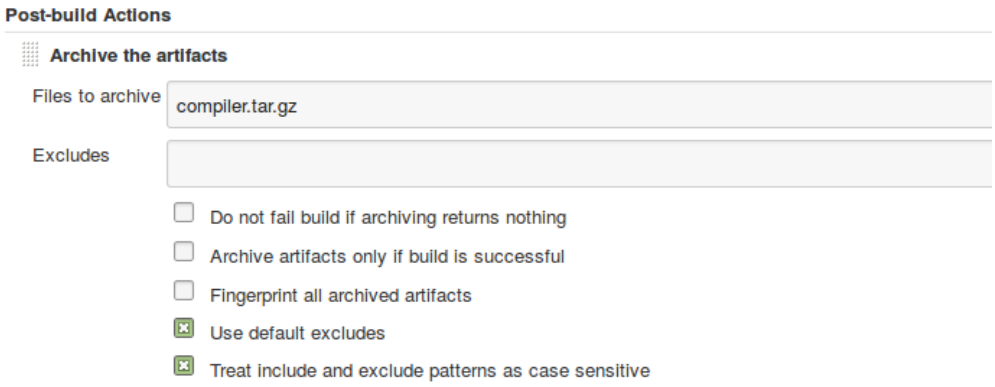
Tento krok je potřebné přidat jako akci před dokončením běhu projektu a zadat názvy souborů, které se mají archivovat. V tabulce 7.4 je možné vidět, jaké artefakty ze kterých projektů jsou ukládány. Na obrázku 7.4 je znázorněn příklad nastavení v prostředí Jenkins.

Stahování artefaktů

Pokud chceme z některého projektu kopírovat artefakty na určitou pracovní plochu jiného projektu, je nutné uvést v konfiguraci projektu, jež obsahuje daný artefakt, záznam o povolení pro konkrétní projekt si artefakt stáhnout. V projektu Parallel-build-python je nutné

Projekt	Artefakt
Parallel-build-compiler	compiler.tar.gz
Parallel-build-dev	dev.tar.gz
Parallel-build-binutils	binutils.tar.gz
Parallel-build-oprofile	oprofile.tar.gz
Parallel-build-python	python.tar.gz
Parallel-build-pack	**/packages/*.deb, **/packages/*.changes, **/packages/*.exe, **/packages/*.rpm

Tabulka 7.4: Seznam artefaktů v projektech



Obrázek 7.4: Možnosti nastavení ukládání artefaktů

nastavit povolení stahování pro Parallel-build-compiler a Parallel-build-dev. U projektů Parallel-build-compiler, Parallel-build-dev, Parallel-build-binutils, Parallel-build-oprofile a Build-ide je potřeba nastavit povolení pro projekt Parallel-build-pack.

Ke kopírování artefaktů pomocí zásuvného modulu Copy Artifact Plugin určíme projekt s konkrétní konfigurací, ze které bude artefakt kopírován. K tomu nám poslouží vytvořené proměnné stroje, které jsme si popsali výše. Dále musíme také označit, ze kterého konkrétního běhu má být artefakt stažen. Jednotlivá nastavení jsou zobrazena v tabulce 7.5.

7.3 Podmíněný běh

Nyní můžeme vytvořit potřebné Hooky k repozitářům python, gnu-binutils a oprofile, abychom zajistili, že se projekty Parallel-build-python, Parallel-build-binutils a Parallel-build-oprofile budou spouštět pouze při detekci změny v těchto repozitářích. Výsledná podoba hooků je znázorněna níže. Nakonec je nutné u těchto projektů zaškrtnout políčko `Poll SCM`, abychom aktivovali detekci hooků a projekty se automaticky spouštěly.

Repozitář	Hook
gnu-binutils	https://jenkins.codasip.com/jenkins/git/notifyCommit?url=ssh://git@codasip3/codasip-framework/gnu-binutils.git
python	https://jenkins.codasip.com/jenkins/git/notifyCommit?url=ssh://git@codasip3/codasip-framework/python.git
oprofile	https://jenkins.codasip.com/jenkins/git/notifyCommit?url=ssh://git@codasip3/codasip-framework/3rd-party-tools

Projekt	**Umístění artefaktu**	**Soubory ke kopírování**	**Běh**
Parallel-build-compiler	Parallel-build-python/r=$SPECIFIER_PYTHON	python.tar.gz	Poslední úspěšný
Parallel-build-dev	Parallel-build-python/r=$SPECIFIER_PYTHON	python.tar.gz	Poslední úspěšný
Parallel-build-pack	Parallel-build-compiler/r=$SPECIFIER_COMPILER	compiler.tar.gz	Ten, jež aktivoval tento projekt
Parallel-build-pack	Parallel-build-dev/r=$SPECIFIER_DEV	dev.tar.gz	Ten, jež aktivoval tento projekt
Parallel-build-pack	Parallel-build-binutils/r=$SPECIFIER_BTLS	binutils.tar.gz	Ten, jež aktivoval tento projekt
Parallel-build-pack	Parallel-build-compiler/r=$SPECIFIER_OPROFILE	oprofile.tar.gz	Ten, jež aktivoval tento projekt
Parallel-build-pack	Build-ide	ide.product*.zip	Určuje proměnná BUILD-SELECTOR

Tabulka 7.5: Nastavení stahování artefaktů

7.4 Spouštění skriptů

Poté, co jsme v prostředí Jenkins vytvořili všechny projekty, můžeme v nich nastavit spouštění potřebných shellových skriptů. Jedná se o soubory, napsané pro příkazový interpret Shell. Tyto skripty jsem implementovala tak, aby zajistili spouštění dalších sad skriptů již implementovaných firmou Codasip, ve kterých je prováděn samotný překlad, instalace a tvorba balíku. V implementovaných souborech bylo nutné také zajistit zpracování argumentů předaných z prostředí Jenkins, dále nastavit proměnné prostředí potřebné k překladu a instalaci, provést rozbalení stažených artefaktů na pracovní plochu projektu a také archivaci instalovaných částí či výsledného balíku. V tabulce 7.6 je znázorněno, jaké skripty, ze kterých projektů v prostředí Jenkins nastavíme ke spuštění.

Projekt	**Skript**	**Popis činnosti**
Parallel-build-compiler	build-compiler.sh install-compiler.sh	Překlad části Compiler Instalace části Compiler
Parallel-build-dev	build-dev.sh install-dev.sh	Překlad části Dev Instalace částí Dev, QEMU, LMX
Parallel-build-binutils	build-binutils.sh install-bintuils.sh	Překlad části GNU-Binutils Instalace části GNU-Binutils
Parallel-build-oprofile	build-oprofile.sh install-oprofile.sh	Překlad části OProfile Instalace části OProfile
Parallel-build-python	build-python.sh	Překlad části Python
Parallel-build-pack	pack-all.sh	Vytvoření balíku rpm/deb/exe

Tabulka 7.6: Implementované skripty nastavené ke spuštění v prostředí Jenkins

Kapitola 8

Testování

Vytvořená hierarchie projektů v prostředí Jenkins spolu se všemi implementovanými skripty a nastaveními byla použita k sestavení plnohodnotného balíku. Tvorba v prostředí Jenkins proběhla úspěšně na obou testovaných konfiguracích (64bitový systém CentOS 6.7 a 64bitový systém Windows 8 pro typ Visual Studio). Následně bylo provedeno porovnání veškerých souborů, které obsahoval původní balík s aktuálně vytvořeným. Verze se vzájemně shodovaly. Balíky se dále instalovaly na oba typy systémů a nevykazovaly žádné problémy při testování své funkčnosti, včetně práce s exportovanými nástroji, grafickým prostředím apod. Následně bylo provedeno jejich úspěšné odinstalování.

8.1 Porovnání výsledků s existujícím řešením

V této sekci je porovnán původní koncept sestavení s nově implementovaným z hlediska délky tvorby balíku. Vzhledem k tomu, že běh projektů Parallel-build-dev a Parallel-build-compiler je paralelizován a je nutné aby byly pro tvorbu balíku dokončeny obě části, závisí nyní doba sestavení zejména na tom, který z těchto projektů poběží déle. Projekty Parallel-build-binutils, Parallel-build-oprofile a Parallel-build-python dobu sestavení balíku ovlivní minimálně. Z těchto projektů se vždy k sestavení použije poslední vytvořený artefakt, který se pouze stáhne na pracovní plochu projeku Parallel-build-dev, Parallel-build-compiler či Parallel-build-pack. Délka sestavení se tak navýší jen o dobu stahování artefaktu. Běh projektu Parallel-build-pack, kde probíhá konečná tvorba balíku, je časově téměř srovnatelný s původní tvorbou. Doba běhu je jen mírně zvýšena kvůli potřebě stažení veškerých artefaktů, jež musí být do balíku umístěny. V tabulce 8.1 jsou zobrazeny implementované projekty, jež ovlivňují dobu procesu sestavení.

Systém	**Projekt**	
	Parallel-build-compiler	Parallel-build-dev
CentOS-6.7-64	00:46:24	00:34:22
Win-8-64-VS	01:01:29	00:45:59
	Parallel-build-pack	
CentOS-6.7-64	00:21:01	
Win-8-64-VS	00:26:41	

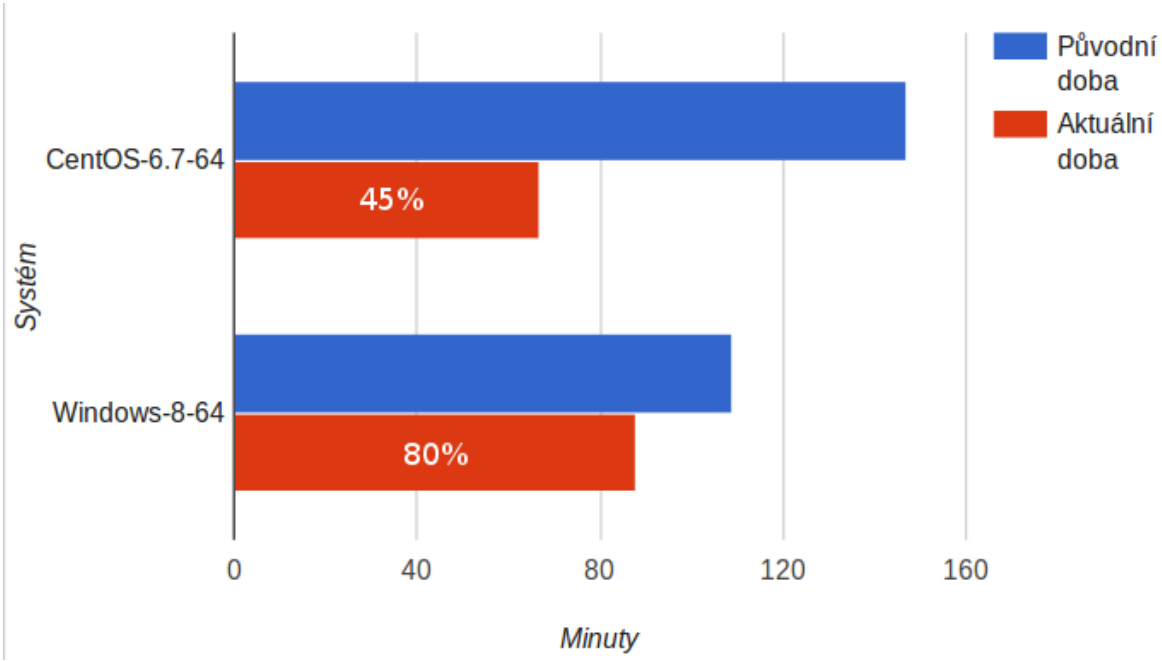
Tabulka 8.1: Doba běhu projektů

Z tabulky vyplývá, že projekt Parallel-build-dev běžící souběžně s projektem Parallel-build-compiler je dokončen jako první a proto jej nemusíme do doby běhu uvažovat. Celková doba sestavení se tedy rovná délce běhu projektu Parallel-build-compiler a projektu Parallel-build-pack. V tabulce 8.2 je porovnána doba potřebná k původnímu způsobu sestavení balíku s dobou, kterou zaujímá nově implementovaný způsob tvorby.

	Část	**Uzel**	**Doba** (h:m:s)
Původní průběh	Překlad	CentOS-6.7-64	02:11:38
		Win-8-64-VS	01:27:09
	Instalace	CentOS-6.7-64	00:01:36
		Win-8-64-VS	00:05:28
	Tvorba balíku	CentOS-6.7-64	00:14:30
		Win-8-64-VS	00:15:43
	Celkem	**CentOS-6.7-64**	**02:27:44**
		Win-8-64-VS	**01:49:20**
Nová implementace	Parallel-build-compiler	CentOS-6.7-64	00:46:24
		Win-8-VS-64	01:01:29
	Parallel-build-pack	CentOS-6.7-64	00:21:01
		Win-8-VS-64	00:26:41
	Celkem	**CentOS-6.7-64**	**01:07:25**
		Win-8-64-VS	**01:28:10**

Tabulka 8.2: Porovnání původní implementace s aktuální z hlediska doby běhu

Na následujícím grafu číslo 8.1 je znázorněno procentuální zrychlení jednotlivých konfigurací oproti původní implementaci.



Obrázek 8.1: Zrychlení procesu sestavení

V případě 64bitového systému CentOS se podařilo dobu sestavení zkrátit přibližně o 1 hodinu a 20 minut. U systému 64bitového Windows doba sestavení klesla přibližně o 21 minut. U systému CentOS se tak podařilo zrychlit proces sestavení o 55 %, u systému Windows jsme dosáhli výsledku 20 %.

Důvod, proč se pro systém Windows nepodařilo získat tak znatelný rozdíl oproti původní implemetaci, má dvě opodstatnění. Prvním z nich je to, že nebylo možné na tomto systému využít paralelní kompilace samotných částí, jelikož implementace kódů firmy Codasip zatím podporuje u platformy Windows pouze jednoprocesorový překlad. Druhým důvodem je to, že části, jež se podařilo paralelizovat (Compiler a Dev) nejsou na systému Windows tak časově náročné jako na systému CentOS. To stejné platí i pro části, jejichž spuštění je závislé pouze na změnách v repozitářích, které jsou danými částmi využívány (OProfile, Python, GNU-Binutils).

Sestavené balíky se však dále používají k navazujícímu automatizovanému testování, které je zahájeno až po vytvoření balíků na všech konfiguracích. Jako celkové zrychlení tak můžeme uvažovat rozdíl mezi dobou, jež trvalo nejdelší sestavení některé z konfigurací v původním principu tvorby balíku a dobou, jež zaujímá aktuální nejdelší sestavení. Uvažujeme-li testované konfigurace, tak se doba tvorby balíků zkrátila z času 2:27:44 na čas 1:28:10. Dosáhli jsme tak celkově 40% zrychlení sestavení.

Kapitola 9

Závěr

Cílem této práce bylo navrhnout a implementovat souběžné sestavení balíku Codasip Studio v prostředí Jenkins. Bylo nutné se zaměřit na problematiku paralelních a distribuovaných výpočtů, objasnit principy průběžné integrace a také nalézt možnosti, jež poskytuje prostředí Jenkins pro souběžné zpracování více úloh. Dále bylo potřeba porozumnět aktuálnímu průběhu sestavování balíku, aby bylo možné tento proces rozdělit na několik menších celků a pokusit se jejich běh paralelizovat.

K souběžnému sestavení bylo v prostředí Jenkins vytvořeno sedm projektů, dále jsem využila několika zásuvných modulů, jež podporovaly potřebnou funkcionalitu při paralelizaci. Vzniklo také několik shellových skriptů, jež zahajovaly procesy překladu, instalace a tvorby výsledného balíku.

Při testování na vybraných konfiguracích (64bitový systém Windows 8 a 64bitový systém CentOS 6.7) se mi v porovnání s původním procesem sestavení podařilo prokázat zrychlení až o 55 %, konkrétně na systému CentOS. Z hlediska obou testovaných konfigurací jsem oproti původní době nejdéle trvajícího sestavení dosáhla celkového zrychlení o 40 %.

Vytvořená implementace je nyní náročnější z hlediska paměťového místa, jež vyžadují artefakty uložené v jednotlivých projektech. Při dnešních velikostech diskového prostoru se ale jedná o zanedbatelné položky. To platí také pro zvýšený provoz po síti při kopírování artefaktů, vzhledem k vysokým přenosovým rychlostem, jež jsou v dnešní době standardem. U systému CentOS je nyní využíváno čtyř procesorů namísto původních dvou. Změna v konfiguraci však byla z testovaných systémů provedena pouze u tohoto, jelikož pro systémy Windows zatím není víceprocesorový překlad zdrojových kódů ze strany firmy Codasip podporován.

Implementovaný proces sestavení předpokládá své nasazení ve firmě Codasip. Vytvořená verze nyní nezahrnuje proces tvorby grafického prostředí Codasip IDE, které je momentálně firmou řešeno v samostatném projektu prostředí Jenkins. Sestavení balíku Codasip Studio je zahájeno až po vytvoření grafického prostředí. V dalším vývoji by tedy bylo vhodné zaměřit se na vazbu mezi těmito dvěma částmi a pokusit se zajistit jejich paralelní běh. Další zajímavou myšlenkou do budoucna by byla možnost souběžného zpracování zdrojových kódů u platformy Windows.

Literatura

[1] Codasip: *Codal Language Reference Manual*. online, 2016.

[2] Codasip: *Codasip Studio Technical Reference Manual*. online, 2016.

[3] Coulouris, G.; Dollimore, J.; Kindberg, T.: *Distributed Systems: Concepts and Design*. Addison-Wesley, Čtvrté vydání, 2005, ISBN 978-0-321-26354-4.

[4] Deshpande, N.; Kamalapur, S.: *Distributed Systems*. Technical Publications, 2009, ISBN 978-8184314045.

[5] Donohue, M.: *Join Plugin*. online, 2016.
URL https://wiki.jenkins-ci.org/display/JENKINS/Join+Plugin

[6] Duvall, P.; Matyas, S.; Glover, A.: *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison Wesley, 2007, ISBN 0-321-33638-0.

[7] Ferguson, J.: *Jenkins: The Definitive Guide*. O'Reilly Media, 2011, ISBN 978-1-449-30535-2.

[8] Gebali, F.: *Algorithms and Parallel Computing*. Wiley, 2011, ISBN 978-0-470-90210-3.

[9] Hanáček, P.: *PRL: Paralelní a distribuované algoritmy 1*. online, 2008.
URL https://www.fit.vutbr.cz/study/courses/PDA/private/www/h001.pdf

[10] Harder, A.: *Copy Artifact Plugin*. online, 2016.
URL https://wiki.jenkins-ci.org/display/JENKINS/Copy+Artifact+Plugin

[11] Hethey, J.: *GitLab Repository Management*. Packt Publishing Limited, 2013, ISBN 978-1-78328-179-4.

[12] Kshemkalayani, A. D.; Singhal, M.: *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2011, ISBN 978-0-521-18984-2.

[13] Lonkar, Y.: *JobFunIn Plugin*. online, 2016.
URL https://wiki.jenkins-ci.org/display/JENKINS/JobFanIn+Plugin

[14] Rodriguez, A.: *Git Plugin*. online, 2016.
URL https://wiki.jenkins-ci.org/display/JENKINS/Git+Plugin

[15] Roosta, S. H.: *Parallel Processing and Parallel Algorithms*. Springer-Verlag New York, 1999, ISBN 0-387-98716-9.

[16] Sauce Labs: Testing trends in 2016: A survey of sotfware professionals. Technická zpráva, jan 2016.
URL https://saucelabs.com/resources/white-papers/sauce-labs-state-of-testing-report-2016.pdf

[17] Sommerville, I.: *Software Engineering* . Addison-Wesley, 2004, ISBN 0-321-21026-3.

[18] Vounckx, J.; Azemard, N.; Maurine, P.: *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*. Springer-Verlag Berlin Heidelberg, 2006, ISBN 978-3-540-39094-7.

Přílohy

Seznam příloh

Příloha A

Obsah CD

Přiložené CD obsahuje následující adresáře:

- `doc/` – technická zpráva ve formátu PDF
- `tex/` – zdrojové soubory a obrázky technické zprávy v jazyku LaTeX
- `src/` – zdrojové kódy implementované v rámci této práce
- `README` – důležité informace